CSCE 608 Database systems - Project 2

Sujith Rengan Jayaseelan, 531006283

a. Project overview:

The project aims to implement two key data-structure/ algorithm used in Database systems, and is broadly divided into two dealing with B+ tree implementation and Two-pass Hash based Join operation implementation respectively.

For B+ trees, the project is roughly divided into four parts. First step involves data generation to generate random key values to be stored in the B+ trees with the given constraints. Next, the actual B+ tree data structure is defined implemented. There are two variants of the B+ tree - dense and sparse. Dense trees have non-root nodes filled to the maximum capacity of their order, while the sparse trees only maintain the minimum number of keys at each node.
Using the implementation of B+ trees, we build 4 trees - dense B+ tree of order 13, sparse B+ tree of order 13, dense B+ tree of order 24, sparse B+ tree of order 24. With all four trees constructed, we conduct experiments with the proposed operations involving insertion, deletion, search, range search on all four trees. For insert and delete operations, the value of all the nodes (keys) that got modified is printed - both values before and after the operation are printed. For search, we print result key(s) resulting from the search or range search operation.

For the 2-pass Hash-based Join, the implementation is similar divided into five parts. First step involves data generation, that involves generating random records for relations. Next, we implement the actual data structures and the associated algorithm required to do Hash-based join operations. Specifically, we introduce VirtualDisk and VirtualMemory classes that help to implement the algorithm. VirtualDisk emulates the behavior of a real Disk and provides APIs to the Virtual memory to read and write from it in blocks (implemented by VirtualDiskBlock data structure). Next we pick the hash function used for the join algorithm — Jenkins Hash algorithm (one-at-a-time variant) which is a fairly simple yet uniform hash function for numbers and string keys.
With the hash function ready, we implement the join algorithm. We emulate the whole process of reading the records block by block from the disk, generating buckets based on the hashes, enforcing one-pass join algorithm and completing the join operation and returning the results. The details of the implementation is explained further in section 2. For the experiments, we execute two join operations for the suggested relations and print the join results and the disk I/Os used for the corresponding joins.

b. B+ trees - Implementation:

1. Data Generation:

For generating random records, we assume that each of the records only contains a search key with no other attributes. The search key is an integer between 100,000 and 200,000. The function `generate_keys` function generates 10000 random keys in the range of 10000 and 20000 both inclusive and returns them as a list.

```
def generate_keys(count = 10000, low = 100000, high=200000):
    return random.sample(range(low, high), count)
```

The function is fairly straightforward as it uses random sampling to return requested count of numbers in the specified range.

2. Building B+ trees:

The core B+ trees is implemented as a BPlusTree class which internally uses BPlusTreeNode class to represent the nodes in the tree. A node in the B+ tree is represented as below.

```
class BPlusTreeNode:
    def __init__(self, threshold, _is_leaf=False):
        self.threshold = threshold
        self.keys = []
        self.children = []
        self.is_leaf = _is_leaf
        self.next = None
        self.prev = None
        self.parent = None
        self.is_root = False
```

The is_leaf makes if the node is a leaf. Similarly is_root denotes if the node is the root of the tree. The threshold attribute controls when the node is split/merged based on the order of the B+ tree. This is also the primary attribute controlling how dense and sparse trees differ. The keys store the key values. The children attributes store the pointers to the children BPlusTreeNode objects bounded by the corresponding key. The next and prev pointers are valid only for leaf nodes.

B+ Tree is constructed using the above node objects. Below is the definition of the tree class.

```
class BPlusTree:
    def __init__(self, order, is_sparse=False):
        self.order = order
        self.is_sparse = is_sparse
        self.threshold = math.ceil(order / 2) if self.is_sparse else order
        self.root = BPlusTreeNode(self.threshold, True)
```

The order denotes the order of the tree. is_sparse specifies if the tree is a sparse B+ tree. The threshold is same as explained earlier in the node and is set appropriately

depending on if the tree is dense or sparse. For dense trees, we pass the threshold as the same value as the order of the tree. But for a sparse tree, by definition the tree can hold only the minimum number of node keys which is given by math.ceil(order/2). Finally, we create a root node for the tree which is initialized with a threshold value and True passed to mark is_root of the node.

We have the build function that builds the B+ tree. Here, we implement build as just a series of inserts. The build function takes a list of keys, and inserts the money by one.

```
def build(self, keys):
      for key in keys:
            self.insert(key)
```

Using the above function, we build 4 trees as follows:

```
def build_trees():
    dense_tree_13 = BPlusTree(13, is_sparse=False)
    dense_tree_13.build(keys)

    sparse_tree_13 = BPlusTree(13, is_sparse=True)
    sparse_tree_13.build(keys)

    dense_tree_24 = BPlusTree(24, is_sparse=False)
    dense_tree_24.build(keys)

    sparse_tree_24 = BPlusTree(24, is_sparse=True)
    sparse_tree_24.build(keys)
```

3. Operations on B+ trees:

We implement 4 operations for the B+ trees: Insert, Delete, Search, Range Search. The following goes over the respective implementations.

Insert:

The `insert` operation implementation is used to insert a new key into the B+ tree data structure. The method takes a key value as input and then calls either the `_insert_into_leaf` method if the node is a leaf node or the `_insert_into_internal` method if the node is an internal node.

The `_insert_into_leaf` method first checks whether the key already exists in the node. If the key already exists, then the method returns `(None, None)` to indicate that the key was not inserted. If the key does not exist, then it is appended to the keys list of the node and the list is sorted. If the length of the keys list is less than or equal to the threshold of the B+ tree, then the method returns `(None, None)` to indicate that the key was inserted successfully. Otherwise, the method calls the `_split_leaf` method to split the node.

The `_insert_into_internal` method calls itself recursively on the appropriate child node, based on the key range. The leaf node returns a tuple containing the key to be

inserted in the parent node and the new child node created if the child node was split. If the child node was split, then the key and new child node are inserted into the parent node using the sorted index of the key. If the length of the keys list of the parent node is greater than the threshold, then the parent node is split using the _split_internal method propagating up. The _split_internal method creates a new internal node, moves the second half of the keys and children of the parent node to the new node, and returns the key to be inserted in the parent node and the new node created. The is_sibling method is used to check whether a given node is a sibling of the current node, based on their parent node. This method returns True if the given node is a sibling of the current node, and False otherwise.

Delete:
The `delete` operation in the B+ Tree takes a key as input parameter. The method calls the `_delete_from_leaf` or `_delete_from_internal` method depending on whether the node is a leaf or an internal node. If the node is a leaf node, the _delete_from_leaf method is called. It checks if the key exists in the node's key list. If the key does not exist in the node's key list, the method returns (None, None) indicating that the key was not found. If the key exists in the node's key list, the method removes the key from the list. If the number of keys in the node is greater than or equal to the minimum threshold, or the node is the root node, the method returns (None, None) indicating that the deletion was successful and there was no need for redistribution or merging. Otherwise, the method tries to redistribute or merge the node with its siblings to maintain the B+ tree properties.

The `_delete_from_leaf` method first tries to borrow a key from its next or previous sibling if either of them is a sibling and has more than the minimum number of keys. If the method borrows a key from the next sibling, it returns (0, self.next.keys[0]), where 0 indicates that the sibling is the next sibling and self.next.keys[0] is the key that was borrowed. If the method borrows a key from the previous sibling, it returns (1, self.keys[0]). However, If the method does not borrow a key from any of the siblings, it tries to merge the node with its next or previous sibling. If the method merges the node with its next sibling, it returns (-1, self.keys[0]), where -1 indicates that the node was merged with the next sibling and self.keys[0] is the new minimum key of the merged node. If the method merges the node with its previous sibling, it returns (-2, self.prev.keys[0]).

The `_delete_from_internal` method finds the child node that should contain the key and calls its _delete method recursively. The _delete method returns the direction of the minimum key of the child node, if the child node has to be merged with its sibling, and the minimum key of the child node if a key was deleted from it. The _delete_from_internal method updates the minimum key of the child node in its key list and returns (None, None) if no further redistribution or merging is required. If redistribution or merging is required, the _delete_from_internal method tries to redistribute or merge the node with its siblings similar to the _delete_from_leaf method.

Finally, if the node is the root and has no keys remaining after a deletion, it sets its first child as the new root of the tree and returns (None, None).

Search:
The `search` function is a recursive method that searches for a key in the B+ tree. If the node is a leaf, it checks if the key is in its keys list and returns the key if it is found, or an empty response if it is not found. If the node is not a leaf, it iterates through the keys of the node and compares the search key with each key. If the search key is less than a key, the method continues the search in the corresponding child node. If the search key is greater than all keys, the method continues the search in the rightmost child node. This is because the keys list of an internal node contains the boundaries between its child nodes, and the rightmost child node is the one whose keys are all greater than the keys in the node.

Range Search:
The `range_search` function takes two arguments start and end which define the range of keys to search. If the current node is a leaf, we perform the search on this node. We iterate over the all the leaf keys starting from the current node and check if the key is within the range defined by start and end. We use the next pointer of the leaf node to move to the left node of the next sibling directly instead of traversing the whole tree again. WE collect all keys that fall in the requested range in a list. If a key is found that is greater than end, we can stop searching since all the keys after that would be greater as well. We return the list of keys that we found within the range. If the current node is not a leaf, we need to traverse down the appropriate child node to find the leaf nodes that contain the keys within the given range. We iterate over the keys of the node to find the appropriate child node to traverse down to continue the search. If the given start value is less than the key of the current node, we can traverse down to the child node to the left of the key. If we have iterated over all the keys and the given start value is greater than or equal to the last key in the current node, we can traverse down to the rightmost child node. Once we have found the appropriate child node to traverse down to, we call the range_search function recursively on that node. We repeat the same process until we reach the leaf nodes and find all the keys within the range. Finally, the function returns the list of keys that are within the range defined by start and end. If no keys are found within the range, an empty list is returned.

c. Two-pass Hash based Join - Implementation:

1. Data Generation:

For generating the relation S(B, C) which we will denote as relation BC for convenience, We assume that each record contains B — which is an integer key (unique) that lies in the range of 10000 and 50000. The attribute C is a randomly generated string of length 10. The function `generateRandomWord` generates a random word for the attribute C. The `generateRandomKey` function returns a random value for attribute B.

```
def generateRandomWord():
    return ''.join(random.choice(string.ascii_lowercase + string.digits) for
_ in range(10))

def generateRandomKey(low, high):
    return random.randint(low, high)

def generateRelationBC(disk, size=5000):
    base_address = disk.getWriteCursor()
    ref, refKeys = [], set()
    for off in range(size//disk.BLOCK_SIZE):
        block = VirtualDiskBlock(disk.BLOCK_SIZE)
        for blk in range(disk.BLOCK_SIZE):
            while True:
                B = generateRandomKey(10000, 50000)
                if B not in refKeys:
                    refKeys.add(B)
                    break
            C = generateRandomWord()
            ref.append((B, C))
            block.data[blk] = (B, C)
        disk.writeBlockSeq(block)
    disk.bucket_base += size//disk.BLOCK_SIZE
    return Relation("BC", base_address, size, ref, refKeys)
```

The above function `generateRelationBC` takes disk object as an input. The main loop generates B and C values for 5000 times. To note here is that we keep generating until a unique B is returned (the existing B keys are stored in a reference set to track). The complete relation is stored in disk. Similar to the above method we also have two helper methods `generateRelationAB` and `generateRelationAB2` that takes care of randomly generating relations AB for experiments. They differ in a small way in that one picks B keys from relation BC records, while the other randomly generates B keys in the range of 20000 to 30000.

2. VirtualDisk and VirtualMemory:

The two main data structures used for implementing hash-based join is the Virtual Disk and the Virtual Memory. In this section we go over how they are implemented.

`VirtualDiskBlock` is a class that represents a single block of data in a virtual disk. A block is a contiguous chunk of data of a fixed size, which can be read or written to/ from the virtual disk.

```
class VirtualDiskBlock:
    def __init__(self, block_size, data=[]):
        self.block_size = block_size
        self.data = [None]*block_size
        for i in range(min(block_size, len(data))):
            self.data[i] = data[I]
```

The constructor of VirtualDiskBlock takes two parameters: block_size and data. block_size specifies the size of the block, and data is an optional parameter that specifies the initial data for the block. If data is not provided, the block is initialized with all elements set to None. If data is provided, it is copied into the block's data array, starting from the first element up to the size of the data or the block size.

`VirtualDisk` class is an abstract representation of a disk drive in computer memory. It provides an interface for reading and writing blocks of data to/from the virtual disk. The `BLOCK_SIZE` represents a constant size of a block in the virtual disk. All blocks in the virtual disk have the same size. The `array` is a list that represents the virtual disk. Each element in the list is a `VirtualDiskBlock` object. The `cursor` — an integer that represents the current write cursor of the virtual disk. `io_count` is an integer that represents the number of I/O operations performed on the virtual disk and `bucket_base` which stores the base address from which bucket data is stored.

```
class VirtualDisk:
    def __init__(self):
        self.BLOCK_SIZE = 8
        self.BUCKET_CAP = 200
        self.array = [None] * self.BUCKET_CAP ** 2
        self.cursor = 0
        self.io_count = 0
        self.bucket_base = 0
```

The VirtualDisk class provides several methods:
- `readBlock(block_id)`: returns the VirtualDiskBlock object at the specified index.
- `writeBlockSeq(block)`: writes the specified VirtualDiskBlock object sequentially to the next available block in the virtual disk.
- `writeBlock(block, block_id)`: writes the specified VirtualDiskBlock object to the block with the specified block index.
- `getWriteCursor()`: returns the current write cursor of the virtual disk.

The VirtualMemory class emulates operations of a memory and in our case primarily focuses on reading and writing to disk. The class represents a virtual memory with a fixed size (in blocks) represented with the `SIZE` attribute and is set to 15 in our case and provides basic read and write operations. It has an `array` attribute that represents the memory itself, as well as a `base_address` attribute that serves as a reference point for all memory addresses. Additionally, it has a `cache` attribute that stores some metadata about the disk addresses of relations and their buckets during join operation.

The VirtualMemory class also provides several interfaces for reading from and writing to the virtual disk. The writeToDiskSeq method writes a block of memory sequentially

to the next available block on the disk, while the writeToDiskLoc method writes a block of memory to a specific block ID on the disk. The readFromDisk method reads a block of memory from a specified block ID on the disk and stores it in the virtual memory. Finally, the flush method can be used to clear the entire virtual memory, setting all values to None.

```
class VirtualMemory:
    def __init__(self):
        self.SIZE = 15
        self.array = [None] * self.SIZE * 8
        self.base_address = 0
        self.cache = [None] * 3
```

3. Hash function:

We need to pick a good hash function that uniformly maps the keys (B values of the relation) to hash numbers/ buckets. Considering the B keys are just integers, we go for the simple Jenkins hash (one at a time variant) that is a fairly simple yet uniform hash function for integer and string based keys. Jenkins hash is a non-cryptographic hash function designed for general-purpose hash table lookup, as well as other hash-based applications such as checksums, data fingerprinting, and is known for its simplicity, speed, and good collision resistance properties.
The Jenkins hash algorithm operates by processing the input data in 4-byte (32-bit) chunks, and performing a series of bitwise operations on these chunks to produce a 32-bit hash value. The algorithm uses a combination of bitwise shifts, XOR and addition operations to mix the input data and produce the final hash value.
One of the key advantages of the Jenkins hash algorithm is its excellent collision resistance properties, meaning that it produces very few hash collisions.

4. Join operation:

The code implements the hash-based join algorithm to efficiently join two relations R1 and R2 (In our case relation BC and AB). The algorithm is designed to work with relations that are too large to fit into memory and therefore rely on disk I/O operations to manage data movement. The hash join algorithm consists of two main phases: Generating buckets for the reactions, Performing one-pass join operation on the buckets.

In the first phase, the generateBuckets function is responsible for creating hash buckets for a given relation R and storing them back on disk. The function takes in the following parameters:
- `mem`: An instance of the Memory class representing the main memory.
- `disk`: An instance of the Disk class representing the disk.
- R: An instance of the Relation class representing the relation for which hash buckets need to be generated.
- `num_buckets`: The number of hash buckets to generate.

The function starts by calling the `getRIndex` function to get the index of the relation in the cache. The cursor of the disk is then set to the starting address of the buckets for this relation. Next, an array of size num_buckets is created in the cache for storing the number of tuples in each bucket. This array is initialized to all zeros.

For each block in the relation R, the block is read from disk into the memory. Then, for each tuple in the block, the `jenkinsHash` function is called to calculate the hash value of the key. The hash value is then used to determine which bucket the tuple should be stored in. The tuple is stored in the bucket in memory at the next available slot, and the count of tuples in the corresponding bucket in the cache is incremented. If the number of tuples in the bucket reaches the maximum size of a block, the bucket is written to disk and the tuples in memory are cleared.
Finally, the function loops through all the buckets and writes any remaining tuples to disk. The memory cache is then flushed to ensure all changes are written to disk.

The second phase is the actual join process, which happens in the `hashJoin` function. This function performs a hash join operation between two relations R1 and R2 and takes in four parameters: mem (a Memory object), disk (a Disk object), R1 (a Relation object), and R2 (another Relation object). (In our case R1 = Relation BC, and R2 is relation AB). The first step of the function is to calculate the number of buckets that will be used for the hash table. This is determined by subtracting 1 from the size of the memory. One block of memory is reserved for reading data. So we use a total number of 14 buckets to hash the tuples into.
Next, the function checks if the corresponding buckets for R1 and R2 has been generated. If not, it generates them using the generateBuckets() function. The metrics attribute of each relation object is updated to store the number of I/O operations used for generating the hash table.

The disk cursor is set to the beginning of the hash table, and an empty list called joinResults is initialized to store the tuples that match during the join operation.
The function then iterates through each bucket in the hash table. For each bucket, it reads in the blocks of data for R2 and stores them in memory. The complete bucket of R2 is now stored in memory. Next, the function reads in the blocks of data for R1 one block at a time. For each tuple in the block, it checks if the matching tuple is present in R2. If the matching tuple is found, it is added to the joinResults list.

Finally, the function returns the joinResults list and the total number of I/O operations used for the hash join operation.

d. Experiments

d.1 B+ trees:

a. A random collection of 10000 keys were generated. (Random seed set to 1)
b. Using the generated keys, dense_tree_13, dense_tree_24, sparse_tree_13, sparse_tree_24 were built.

c1. Two Random keys were generated and inserted in each of dense trees. Sample output for each of the insert is given below:

```
[B+ 13 dense] ++Inserting 163215
Leaf node: BEFORE:
[163206, 163238, 163241, 163295, 163299, 163318, 163348]
Leaf node: AFTER:
[163206, 163215, 163238, 163241, 163295, 163299, 163318, 163348]
[B+ 13 dense] --Inserting 163215


[B+ 24 dense] ++Inserting 198698
Leaf node: BEFORE:
[198644, 198663, 198683, 198696, 198724, 198730, 198734, 198740, 198742, 198751, 198758,
198780, 198784, 198800, 198812, 198816, 198818, 198821, 198826, 198827, 198828, 198830,
198837, 198846]
Leaf node: AFTER:
[198644, 198663, 198683, 198696, 198698, 198724, 198730, 198734, 198740, 198742, 198751,
198758, 198780, 198784, 198800, 198812, 198816, 198818, 198821, 198826, 198827, 198828,
198830, 198837, 198846]
Leaf node SPLIT AFTER:
[198644, 198663, 198683, 198696, 198698, 198724, 198730, 198734, 198740, 198742, 198751,
198758]
[198780, 198784, 198800, 198812, 198816, 198818, 198821, 198826, 198827, 198828, 198830,
198837, 198846]
Internal node BEFORE:
[196340, 196478, 196659, 196889, 197097, 197216, 197405, 197536, 197637, 197749, 197864,
197972, 198143, 198293, 198484, 198644, 198847, 199028, 199323, 199483, 199583, 199785]
Internal node AFTER:
[196340, 196478, 196659, 196889, 197097, 197216, 197405, 197536, 197637, 197749, 197864,
197972, 198143, 198293, 198484, 198644, 198780, 198847, 199028, 199323, 199483, 199583,
199785]
[B+ 24 dense] --Inserting 198698
```

We can see in the first insert of 163215, the leaf node contained less than 13 keys after the insert, and hence no further action was required. However, in the insertion of 198698 to the tree with order 24, the node overflows after insertion that leads to the leaf node being split into two. Furthermore, the corresponding parent internal node is updated with the new child to account for the split.

c2. Two existing keys were picked randomly and deleted from each of the sparse trees. Sample output for two of those operations are given below:

```
[B+ 13 sparse] ++Deleting 195267
Leaf node: BEFORE:
[195263, 195264, 195266, 195267, 195286, 195312, 195318]
Leaf node: AFTER:
[195263, 195264, 195266, 195286, 195312, 195318]
[B+ 13 sparse] --Deleting 195267


[B+ 24 sparse] ++Deleting 150379
Leaf node: BEFORE:
[150366, 150370, 150375, 150379, 150407, 150422]
```

```
Leaf node: AFTER:
[150366, 150370, 150375, 150407, 150422]
Leaf node: BORROW BEFORE: SIBLING:
[150436, 150455, 150464, 150466, 150472, 150476, 150509, 150513]
Leaf node: BORROW AFTER: LEAF:
[150366, 150370, 150375, 150407, 150422, 150436]
Leaf node: BORROW AFTER: SIBLING:
[150455, 150464, 150466, 150472, 150476, 150509, 150513]
Internal node: BEFORE:
[150317, 150366, 150436, 150522, 150611, 150638, 150666, 150738, 150884, 150939, 151008,
151078]
Internal node: AFTER:
[150317, 150366, 150455, 150522, 150611, 150638, 150666, 150738, 150884, 150939, 151008,
151078]
[B+ 24 sparse] --Deleting 150379
```

We can see in the first delete of 195267, the leaf node contained at least the minimum number of nodes (4) for a B+ sparse tree of order 13, and hence no further action was required after the key was deleted. However, in the deletion of 150379 from the tree with order 24, the node now contains only5 keys which is less than the minimum keys (13) required — so the siblings are checked and the node borrows a key from a sibling with sufficient number of keys. The value of the node and its sibling after the borrow is also printed for reference.

c3. Similar to the above two experiments, a total of 5 iteration and deletion operations were run on each of the B+ trees and their output was printed.
To analyse the time complexity of these insert and delete operations, if the order of the tree is denoted by ‘k’, and the total number of elements in the tree is denoted by ’n’, then the average insert operation takes $O(k*\log_k(n))$ where $\log_k(n)$ denotes to traverse the height of the tree to reach the leaf node, while at each node we also iterate through the keys to find which child to go down with takes k operations.
Similarly, the delete operation is very similar and takes $O(k*\log_k(n))$ time in average.

c4. Total of 5 search and range search each were applied to each of the trees. For search, a random key was generated to search. For range search a random lower bound was generated to search. The search results were printed. Below are a few sample search operations.

```
[B+ 24 sparse] ++Searching 125830
FOUND KEY: [125830]
[B+ 24 sparse] --Searching 125830

[B+ 13 dense] ++RangeSearching (195887, 195913)
Leaf nodes iterated: FOUND 5 KEYS: [195901, 195903, 195904, 195909, 195912]
[B+ 13 dense] --RangeSearching (195887, 195913)

[B+ 24 sparse] ++RangeSearching (156616, 156656)
Leaf nodes iterated: FOUND 5 KEYS: [156617, 156621, 156634, 156639, 156652]
[B+ 24 sparse] --RangeSearching (156616, 156656)

[B+ 13 dense] ++Searching 130861
KEY NOT FOUND
[B+ 13 dense] --Searching 130861
```

To analyse the time complexity of these search operation, if the order of the tree is denoted by ‘k’, and the total number of elements in the tree is denoted by ’n’, then the average search operation takes $O(k*\log_k(n))$ where $\log_k(n)$ denotes to traverse the height

of the tree to reach the leaf node, while at each node we also iterate through the keys to find which child to go down with takes ‘k’ operations.
Similarly, the range search operation takes $O(k*\log_k(n) + r)$ time in the worst case. Where ‘r’ denotes the range of the keys to be searched

d.2 Hash based Join:

1. Join of BC and AB. We have 5000 tuples of Relation BC and 1000 tuples of AB with keys picked from keys of BC with duplicates allowed. To print the output, 20 random B keys are picked form AB and the results with those keys are printed. The output is as follows:

```
Join verified for BC and AB. Total tuples: 1000
Total disk IO: 2282

BC ⋈ AB
    B     C             A
-----  ----------  ----------
23906  en49do1zc9  8mslc73svz
18448  81mo7ucuja  809bsoloje
14282  rkij9btmnk  nihc7opbrv
36864  ol2p6nf4t3  g2lefzis78
21237  atztfoa7f3  ob6l2vo5qc
35369  2x5643qvnq  wzf6kvdh3c
14759  5gi1jungcx  ywtr1e4x9y
14759  5gi1jungcx  07ceyve4fa
29882  smn1qpeanp  jg0ioeu7xt
41318  z3szyt74ah  2w4aekm52k
44561  h8restuob6  6mluexb8rd
34986  meqcias1bp  cvpt1erbso
16542  fqzb01uz3j  hkizcw1bhc
31798  4llt9j7jso  6lntnt1uut
21482  bw1qa4zdiy  q9nzgtvrru
17665  8tny4w8enn  j5cn6vsj0z
43563  suio0lxk90  gvr6k9i7tt
22495  2u89ostpja  i0y4wrkfyz
22495  2u89ostpja  qcdqi52rlg
15123  77qam5jqho  yojbvtc6yw
38975  fcjqfmvmtd  10qjueu31n
19048  f1w8qmn9l5  n461futy9x
```

In the above experiment the total blocks used by BC is 5000/8 = B(BC) = 625. Similarly total blocks used by relation AB = 1000/8 = B(AB) = 125. As discussed in class, and reviewing the implementation of the join operation, we use 2*(B(BC) + B(AB)) for generating buckets as we read the complete relation from the disk and write back all the buckets back to disk. Next for the one-pass hash join algorithm we read the AB relation bucket by bucket and BC block by block, but read the complete relation regardless which counts for B(BC)+B(AB) I/O operations.
Totally we have 3*(B(BC) + B(AB)) disk I/O operations. Here this roughly estimates to 3 * 750 = 2250 operations. As we can see in the above output, we have 2282 (~2250) operations. To note that these additional disk IO operations are incurred due to uneven bucket entries requiring additional operations to write and read them from disk.

The total tuples in the join is 1000. This is obvious as the relation AB contains keys from BC, so every element in AB satisfies join with the corresponding key in BC. Randomly picked 20 B keys and the corresponding join results are also printed above.

2. Join of BC and AB2. We have 5000 tuples of Relation BC and 1200 tuples of AB with randomly generated keys in the range of 20000 to 30000. Here we print the output of all the join results. The output is as follows:

```
Join verified for BC and AB2. Total tuples: 152
Disk IO (BC buckets pre-computed): 1097
Total Disk IO (recounting BC bucket generation): 2355

BC ⋈ AB2
    B  C           A
-----  ----------  ----------
25133  i6n8mwf1bn  tczxpzzi9a
20682  xfsfd7wwxh  0s7ygkun46
20609  z9pvodxk2o  kz7tnikwu4
28978  nopkslkwx5  1enqgp7zqg
26445  l68sdcm0oj  bwsraz91hv
26166  qdludzevpc  1osu3rae7n
26871  8jv8ml0de4  ehut5foeew
20144  oym0ul5lmw  6ichitg2al
27501  1d3uvboa7k  gcwy5j5faw
28338  p3rk9pqxmk  cp8hw18u86
26749  mzkla481yj  yp25la7qmb

. . .

29572  utbhxapzc9  k15xnrk6m9
21288  2j64gn8t4b  8z6t811d4i
28356  in0jhu8uml  cwogwfdi3f
25602  jobuca8bfs  5gjmcogc6g
23645  kyee38311p  z0d5ky7jm4
```

In the above experiment the total blocks used by BC is 5000/8 = B(BC) = 625. Similarly total blocks used by relation AB = 1200/8 = B(AB) = 150. Since we already have buckets of BC in disk we only have B(AB)*2 for generating buckets as we read the complete relation from the disk and write back all the buckets back to disk. Next for the one-pass hash join algorithm we read the AB relation bucket by bucket and BC block by block, but read the complete relation regardless which counts for B(BC) +B(AB) I/O operations.
Totally we have B(BC)+ 3*B(AB) disk I/O operations.
Here this roughly estimates to 3 * 150 + 625 = 1075 operations. As we can see in the above output, we have 1097 (~1075) operations. To note that these additional disk IO operations are incurred due to uneven bucket entries requiring additional operations to write and read them from disk. We also print the projected total disk IO if relation BC was not pre-computed on the disk and that would have been roughly 2355 disk IO operations.
The total tuples in the join is 152. We print all the 152 tuples in the output. For simplicity I have only pasted the beginning and the end of the output above.

e. Discussion:

The project was a lot of fun to work on. To implement the algorithm and data structures reviewed in class and actually visualize how they work in real-life systems, was a great learning experience. An early challenge was to effectively design the classes required to implement the algorithms. Specifically I had to carefully consider the design of VirtualDisk and VirtualDiskBlock to make sure I emulate the working of the disk in the best way possible. For B+ trees, the design was a lightly straightforward than the Hash Join implementation as it involved only two broad classes, the node and the tree respectively.
For the B+ trees, I had to carefully consider all the different edge cases for insertion and deletion. I also went ahead and implemented a verify search function that validates the search result. For the hash based join implementation, apart from the design of the disk and memory classes, the actual memory management to read data from the disk and store all bucket buffers was challenging to initially setup. However, prototyping on a smaller relation and memory size helped me to code the implementation and later test it on the requested experiment. Another major challenge was debugging with a large dataset. Both B+ tree (dense trees with order 24 especially) and the hash join experiments had very large datasets with large ranges, which was hard to debug with typical print statements. It was not always easy to identify and isolate errors. I had to resort to debugger tools (PyCharm) to run the programs step by step and identify the issues.

Overall, this project gave me a better and deeper understanding of the algorithms I implemented. I learned how B+ trees are implemented and how the insert, delete, search operations work and how they improve performance in real databases. Moreover, I also gained a deeper appreciation for the challenges of working with large datasets and the importance of debugging tools.

Project source code:

Source code — https://github.com/sujithrengan/dbms-ds-608

Please install tabulate by running **`python3 -m pip install tabulate`**

```
cd dbms-ds-608/

python3 bplus_tree.py

python3 hash_join.py
```